Klausur Informatik 2
10. April 2001

Biedl / Lemke
Schöner / Tschirley

Name: ....

Vorname: ....

Matr.-Nr.: ....

Bearbeitungszeit: 120 Minuten

- Benutzen Sie für die Lösung der Aufgaben nur das mit diesem Deckblatt ausgeteilte Papier. **Lösungen, die auf anderem Papier geschrieben werden, können nicht gewertet werden!**
- Schreiben Sie Ihre Lösungen auf das Aufgabenblatt der jeweiligen Aufgabe. Verwenden Sie auch die Rückseite. **Schreiben Sie keine Lösungen einer Aufgabe auf ein Blatt, das nicht zu dieser Aufgabe gehört!** Wenn Sie zusätzliche, von uns ausgegebene Blätter verwenden, geben Sie unbedingt an, zu welcher Aufgabe die Lösung gehört.
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen können nicht gewertet werden.
- Schreiben Sie nur in **blau** oder **schwarz**. Schreiben Sie nicht mit Bleistift!
- Bei der Implementation von OPAL-Funktionen brauchen *keine* Importe mit angegeben zu werden, wenn Funktionen aus dem gegebenen Teil der Bibliotheca Opalica (oder aus der Struktur `Nat`, `Real`, `Seq` oder den Strukturen für die Listenfunktionale) oder sonstigen (in den Aufgaben) gegebenen Strukturen benutzt werden!
- Die OPAL-Implementierungen müssen nicht dokumentiert werden.
- Erscheint Ihnen eine Aufgabe mehrdeutig, wenden Sie sich an die Betreuer.
- Tragen Sie jetzt (vor Beginn der Bearbeitungszeit) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.

Aufgabe	Punkte	erreicht
1	15	
2	8	
3	10	
4	14	
5	3	

1. Aufgabe (15 Punkte): Imperative Programmierung

1.1. Umsetzen rekursiver in iterative Programme (6 Punkte)

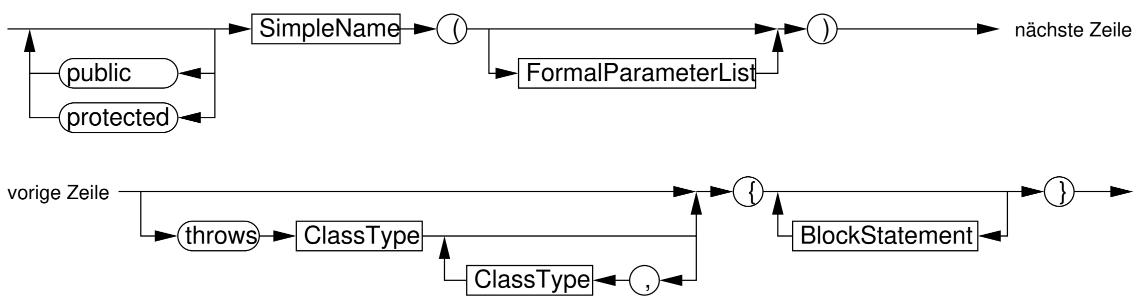
Setzen Sie die folgende OPAL-Funktion in *zwei* verschiedene Java-Methoden um. Die eine soll rekursiv formuliert sein, die zweite iterativ (mit einer Schleife anstelle der Rekursion). Beide sollen sich so eng wie möglich an der gegebenen OPAL-Funktion orientieren.

```
FUN quersumme: nat -> nat
DEF quersumme == \\n.
  IF n=0 THEN 0
  IF n>0 THEN mod(n,10) + quersumme(div(n,10))
  FI
```

1.2. Syntaxdiagramme und EBNF-Grammatiken (3 Punkte)

Transformieren Sie das im folgenden gegebene Syntaxdiagramm in eine EBNF-Grammatik.

Beispielsyntax:



1.3. Heaps mit beliebigem Grad (6 Punkte)

Als Generalisierung von binären Heaps (Heaps mit dem Grad 2) kann man *Heaps mit dem Grad* d aufbauen. Bei diesen Heaps hat jeder Knoten nicht 2 Kind-Knoten, sondern d Stück. Wird ein solcher Heap in einem Array gespeichert, so berechnen sich die Indizes der Kind-Knoten eines gegebenen Knotens i $\in [0, \ldots, N]$ als: d*i + 1, ..., d*i + d. Schreiben Sie eine Java-Methode, die einen Heap des Typs Heap und einen Integer erhält, um die Integer-Zahl in den Heap einzufügen (und die Heap-Bedingung wieder herzustellen).

```
public class Heap {
    int[] data;     // Array, in dem der Heap abgelegt ist
    int heapSize;   // Anzahl der Elemente in 'data', die vom Heap belegt werden
    int d;          // Grad des Heaps
}
```

2. Aufgabe (8 Punkte): Aufwandsanalyse

2.1. Aufwandsklassen (3 Punkte)

Gegeben sei die folgende Behauptung: Für zwei beliebige Funktionen $f(x)$ und $g(x)$ ist es immer der Fall, dass mindestens eine der Aussagen $f \in \mathcal{O}(g)$ oder $g \in \mathcal{O}(f)$ zutrifft. D.h. es gibt keine Funktionen f und g, für die $f \notin \mathcal{O}(g) \wedge g \notin \mathcal{O}(f)$. Trifft diese Behauptung zu? Begründen Sie Ihre Antwort (z.B. indem Sie ein Gegenbeispiel angeben).

2.2. Aufwandsanalyse eines Programmfragments (5 Punkte)

Bestimmen Sie die Funktion für den zeitlichen Aufwand der folgenden Methode power2 im Worst-Case. Leiten Sie daraus mit Hilfe des $\mathcal{O}$-Kalküls oder der im Skript oder unten angegebenen Regeln eine Aufwandsklasse ab. Nehmen Sie einen Zeitaufwand von einer Einheit für jeden Operator, jeden Funktionsaufruf und jedes return an. Achten Sie auf die Nachvollziehbarkeit Ihres Vorgehens; insbesondere müssen die gezählten Operatoren, etc. eindeutig einzelnen Zeilen der Methode zugeordnet werden können!

```
static int power2(int n) {
    if (n == 0) {
        return (1);
    } else {
        return (power2(n-1) + power2(n-1));
    }
}
```

Mit welchem einfachen Trick ist es möglich, den Aufwand linear zu machen?

Definition ($\mathcal{O}$-Kalkül):
Sind f und g zwei positiv-reellwertige Funktionen über $\mathbb{N}$, dann bedeutet $f \in \mathcal{O}(g)$, dass es eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}$ gibt, sodass für alle $n > n_0$ gilt: $f(n) \leq c * g(n)$. g ist dann eine obere Schranke für f.

Regeln für rekurrente Aufwandsformeln:

Rekurrenzrelation		$\mathcal{O}(A)$
$A(n) = A(n-1) + bn^k$		$\mathcal{O}(n^{k+1})$
$A(n) = cA(n-1) + bn^k$	mit $c > 1$	$\mathcal{O}(c^n)$
$A(n) = cA(n/d) + bn^k$	mit $c > d^k$	$\mathcal{O}(n^{\log_d c})$
$A(n) = cA(n/d) + bn^k$	mit $c < d^k$	$\mathcal{O}(n^k)$
$A(n) = cA(n/d) + bn^k$	mit $c = d^k$	$\mathcal{O}(n^k \log n)$

3. Aufgabe (10 Punkte): Programmverifikation

3.1. Formalisieren von Aussagen (2 Punkte)

Formalisieren Sie die folgenden Aussagen (Hilfsprädikate sind mit zu definieren):

1. Der Array arr erfüllt im Indexbereich von 0 bis n die Heap-Bedingung.
2. x ist das Maximum von a und b. (Benutzen Sie hier *kein* Prädikat max!)

3.2. Entwickeln eines Programms aus einer Spezifikation (4 Punkte)

Gegeben seien die folgenden PRE- und POST-Bedingungen, sowie die Invariante I einer Funktion root(int n), die die ganzzahlige Quadratwurzel einer natürlichen Zahl n bestimmt (in der lokalen Variablen res). Implementieren Sie in Java eine Funktion, die diese Aufgabe erledigt und dabei die angegebene Spezifikation erfüllt.

- PRE: $n \in \mathbb{N}^+$
- POST: $res \in \mathbb{N}^+ \wedge \big(res \cdot res \leq n\big) \wedge \big((res+1) \cdot (res+1) > n\big)$
- I: $res \in \mathbb{N}^+ \wedge \big(res \cdot res \leq n\big)$

3.3. Partielle Korrektheit (4 Punkte)

Führen Sie den angefangenen Beweis der partiellen Korrektheit für die folgende Funktion zu Ende. Verwenden Sie dafür den Vordruck auf der nächsten Seite.

```
static double heaviside(double x) {
    int res;
    // PRE: true
    if (x < 0) {
        res = 0.0;
    } else {
        res = 1.0;
    }
    // POST: (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    return(res);
}
```

```
static double heaviside(double x) {
    int res;
    // PRE: true

    if (x < 0) {

        res = 0.0;

    } else {

        res = 1.0;

    }

    // POST: (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    return(res);
}
```

4. Aufgabe (14 Punkte): SCRAM-Assemblerprogrammierung

Implementieren Sie ein Assemblerprogramm, das die aus dem Übungsbetrieb bekannte Funktion `toBase` nachbildet. Gehen Sie dazu in folgenden Schritten vor:

4.1. (3 Punkte) Implementieren Sie ein SCRAM-Assemblermakro `MOD`, das die Modulo-Operation auf dem Inhalt des AC mit dem übergebenen Parameter ausführt. Das Ergebnis der Berechnung soll im AC stehen, wenn das Makro beendet wird.

Achten Sie darauf, keine unnötigen Instruktionen zu verwenden. Verwenden Sie zur Lösung einen iterativen Algorithmus, der *nicht* die Befehle `DIV` und `MUL` enthält.

Beispiel:

Wenn vor einer Instruktion der Wert 13 im AC enthalten ist, so wird nach dem Befehl `MOD 5` der AC den Wert 3 enthalten.

4.2. (5 Punkte) Implementieren Sie ein SCRAM-Assemblermakro `POW`, das den Inhalt des AC so oft mit sich selbst multipliziert, wie der übergebene Parameter angibt (es handelt sich hier um die Berechnung des mathematischen Ausdrucks $AC^{Parameter}$). Das Ergebnis der Operation soll im AC hinterlegt werden.

4.3. (6 Punkte) Die Zahl, die konvertiert werden soll, befindet sich an der Speicherstelle mit dem Label `src`. Die Zahl ist im Dezimalsystem codiert.

Implementieren Sie mit Hilfe der beiden oben entworfenen Makros das Hauptprogramm, das die Zahl `src` in ein beliebiges Zahlensystem der Basis `base` mit $base \in [2..10]$ konvertiert. Die Basis befindet sich an der Speicherstelle mit dem Label `base`. Das Ergebnis soll sich nach Abarbeitung des Programms an der Speicherstelle mit dem Label `erg` befinden.

Hinweis: Sie dürfen zur Lösung der Aufgabe die folgenden Makros verwenden:

- **INC** Makro zum Inkrementieren einer Variablen. Der Wert der als Parameter angegebenen Variablen wird um 1 erhöht und sowohl im AC als auch in der Variablen selbst abgelegt. Nicht nutzbar im unmittelbaren Adressierungsmodus.
- **DEC** Makro zum Dekrementieren einer Variablen. Der Wert der als Parameter angegebenen Variablen wird um 1 verringert und sowohl im AC als auch in der Variablen selbst abgelegt. Nicht nutzbar im unmittelbaren Adressierungsmodus.
- **JNS** Makro zum bedingten Sprung bei negativem AC-Inhalt. Als Parameter wird analog zu den anderen bedingten Sprüngen die Sprungmarke angegeben, zu der ggf. gesprungen wird. Ist der AC-Inhalt nicht negativ, wird normal fortgefahren. Der Wert des AC wird nicht verändert.

Wenn Sie weitere Makros verwenden, so ist der Quellcode anzugeben.

5. Aufgabe (3 Punkte): Zahlensysteme und SCRAM

5.1. Zahlensysteme (1 Punkte) Zahlen werden im Computer im b-adischen Zahlensystem dargestellt, wobei die Basis 2 ist. Beantworten Sie die folgenden Fragen. Begründen Sie Ihre Antwort!

1. Ist diese Darstellung der natürlichen Zahlen umfassend?
2. Ist sie eindeutig?

5.2. SCRAM (2 Punkte)

1. Prüfen Sie kurz, ob das `MAR` im SCRAM tatsächlich notwendig ist. Ist es unnötig und kann man es ohne große Probleme entfernen, oder ist seine Anwesenheit zwingend notwendig? Begründen Sie Ihre Antwort!
2. Zählen Sie auf, was nötig ist, um einen Speicherzugriff durchzuführen. Erläutern Sie kurz den Ablauf.